ESPECIFICACIÓN FORMAL EN OCL DE REGLAS DE CONSISTENCIA ENTRE LOS DIAGRAMAS DE CLASES Y CASOS DE USO DE UML Y EL MODELO DE INTERFACES

Carlos Mario Zapata[1], Guillermo González[2]

Recibido: 23/03/2007
Aceptado: 17/03/2008

RESUMEN

En el ciclo de vida del software, durante las fases de definición y análisis, se realiza una especificación de los requisitos. Para ello, es necesario realizar un proceso de captura de las necesidades y expectativas de los interesados, que se traduce posteriormente en un conjunto de modelos que representan tanto el problema como su solución. Por lo general, la mayoría de esos modelos se expresan en el lenguaje de modelado unificado –UML-, que define un conjunto de artefactos que permiten especificar los requisitos del software, los cuales deberían guardar consistencia, cuando se trate del mismo modelo. Sin embargo, la consistencia entre diferentes artefactos no se encuentra definida en la especificación de UML y poco se ha trabajado con este tipo de consistencia.

En este artículo se propone un método para verificar la consistencia entre el diagrama de clases y el diagrama de casos de uso de UML de una manera formal. Dicho proceso se lleva a cabo evaluando una serie de reglas definidas en el lenguaje de restricciones de objetos –OCL- que se deben cumplir para garantizar que la información brindada por dichos modelos sea consistente. Como se reconoce la participación de los dos diagramas en la elaboración de las interfaces gráficas de usuario –GUI-, se define adicionalmente la consistencia con este artefacto.

Palabras clave: UML, reglas de consistencia, OCL, casos de uso, diagrama de clases, interfaces gráficas de usuario, XML, XMI, Xquery.

1 Escuela de Sistemas, Universidad Nacional de Colombia. Carrera 80 N° 65-223, Bloque M8. Medellín-Colombia. Tel: 4255350. E-mail: cmzapata@unalmed.edu.co

2 Escuela de Sistemas, Universidad Nacional de Colombia. Carrera 80 N° 65-223, Bloque M8. Medellín-Colombia. Tel: 4255350E-mail: ggonzal@unalmed.edu.co

FORMAL OCL SPECIFICATION OF CONSISTENCY RULES BETWEEN THE UML CLASS AND THE USE CASE MODELS AND THE INTERFACES MODEL

ABSTRACT

In a software lifetime, during definition and analysis stages, a specification of requirements is carried out. For such a purpose, it is necessary to get through a process to capture interested persons' needs and expectations, which will later be translated into a set of models representing both the problem and the solution. Most models are frequently expressed by the UML (Unified Modeling Language) which defines a set of devices for specifying software requirements which should be consistent with the same model. However, consistency among several devices is not defined in the UML specification and not too much work has been made with this type of consistence.

This article proposes a method to verify consistence among UML class diagram and use case diagram in a formal way. Such a process is carried out through an evaluation of several rules defined in the OCL (Object Constraint Language), which should be fulfilled to assure that information provided by such models is consistent. As both diagrams participation is recognized when preparing GUI (Graphic User Interfaces) consistence with this device is additionally defined

Keywords: UML, consistence rules, OCL, use cases, class diagram, graphic user interfaces, XML, XMI, Xquery.

INTRODUCCIÓN

Un proceso de desarrollo de software tiene como propósito la producción eficaz y eficiente de un producto software que cumpla con las necesidades y expectativas de los interesados o participantes (stakeholders es el término en inglés). Este proceso empieza típicamente cuando se identifica un problema que puede requerir una solución computarizada, cuyo desarrollo requiere una especificación de requisitos que se logra durante las fases de definición y análisis. Esta especificación es a menudo informal y posiblemente vaga, lo que se suele denominar un "bosquejo áspero" (Jackson, 1995). Los ingenieros de requisitos necesitan examinar esta representación escrita, que es frecuentemente incompleta e inconsistente, basados en la información disponible y en su experiencia previa, para transformar este "bosquejo áspero" en una especificación correcta de requisitos. Luego, se presentan dichos requisitos a los interesados para su validación. Como resultado, se identifican nuevos requisitos para ser agregados a la especificación, o algunos de los previamente identificados podrían eliminarse para mejorarla; por tanto, en cada paso del desarrollo de una pieza de software, la especificación puede perder o ganar requisitos. Una de las tareas críticas de los ingenieros de requisitos en este proceso es asegurar que la especificación de requisitos permanezca correcta en cada paso, o que por lo menos los errores se encuentren lo más rápido posible y que sean identificadas sus fuentes para su revisión (Zowghi y Gervasi, 2002).

En general, los métodos de desarrollo de software (por ejemplo el Unified Process (Jacobson, et al., 1999)), son iterativos e incrementales, repitiendo una serie de iteraciones sobre el ciclo de vida de un sistema. Cada iteración consiste de un paso a través de las etapas de requerimientos, análisis, diseño, implementación y prueba. El resultado de cada iteración representa un incremento sobre cada uno de los modelos construidos en las etapas anteriores. Durante estos ciclos de desarrollo, los modelos se construyen sucesivamente en un nivel más y más bajo de abstracción hasta que el nivel requerido para la codificación es alcanzado. El costo de los errores encontrados al principio en el ciclo de desarrollo es muy inferior al costo de los errores encontrados en los últimos pasos tales como codificación o pruebas, por lo que se debe tratar de hacer un buen análisis desde el principio del proceso, especialmente en la especificación de requisitos, para evitar problemas futuros, que requerirían, incluso, un nuevo diseño, perdiendo así gran parte del trabajo realizado hasta ese momento.

Es frecuente el caso en que, en una tentativa por mantener la consistencia dentro de los requisitos, se eliminen uno o más de la especificación, y no se pueda preservar su integridad. Inversamente, cuando se agregan nuevos requisitos a la especificación para hacerla más completa, es posible introducir inconsistencia en ella (Zowghi y Gervasi, 2002). Dado que la especificación de los requisitos se logra con un conjunto de diagramas, que representan vistas interconectadas del modelo del problema, las inconsistencias pueden surgir entre cualquier par de diagramas, y se pueden acentuar en tanto esos artefactos se usan con mayor frecuencia, como es el casos de los diagramas de casos de uso y de clases.

A medida que se utilizan más artefactos, el problema de la consistencia entre ellos aumenta. Muy pocas técnicas de modelamiento de requisitos tienen una aproximación sistemática para tratar el problema de la consistencia intermodelos (Egyed, 2001). El problema de la consistencia es simplemente ignorado (y dejado a los ingenieros de requisitos y a los interesados, que tienen que validar la especificación de estos con procesos muchas veces manuales).

Después de haber reunido los requisitos, se debe hacer un modelamiento del problema para

obtener una visión más detallada del sistema y así poder resolver por medio del computador las diferentes situaciones que se presentan en el problema a tratar. En este proceso, es preferible trabajar con un estándar de modelamiento, que sea compartido y comprendido por los analistas de diferentes sitios. Utilizando las reglas proporcionadas por el estándar, los ingenieros de software pueden crear modelos concretos y sin ambigüedades. Booch et al (1997) han definido un estándar de modelado denominado UML (Unified Modeling Language) (OMG, 2007). UML está conformado por un conjunto de artefactos que permiten especificar los requisitos del software. La forma de representar a UML es conocida como metamodelo de UML, y está disponible al público junto con la definición del estándar en inglés (OMG, 2007). El metamodelo de UML sirve para que los ingenieros de Software puedan verificar la corrección de sus modelos, ya que tiene la estructura y las reglas que definen las interacciones entre los diferentes diagramas.

UML propone, entre otras cosas, un conjunto de diagramas que representan un software desde distintos puntos de vista. Los diagramas UML son independientes pero se conectan; su metamodelo los describe bajo el mismo techo (OMG, 2007). Por ejemplo, un diagrama de casos de uso (Jacobson, 1992) muestra la funcionalidad que ofrece el sistema futuro desde la perspectiva de los usuarios externos al mismo, en tanto que un diagrama de clases (Jacobson, 1992) representa la estructura estática que las clases poseen y un diagrama de interacción (Jacobson, 1992) representa cómo los objetos intercambian mensajes.

Si bien no pertenece al estándar de UML, el modelo de interfaces describe la presentación de información entre los actores y el sistema (Weitzenfeld, 2005) y es complementario con la información que se presenta en los diagramas de clases y casos de uso. En este modelo, se especifica en detalle cómo se verán las interfaces gráficas de usuario al ejecutar cada uno de los casos de uso. Normalmente, un prototipo funcional de requisitos que muestre la manera en que funcionan las interfaces gráficas de usuario posibilita la comprensión del software futuro por parte de los interesados, quienes pueden, incluso, validar si la información que allí se presenta es correcta y adecuada. Esto ayuda al interesado a visualizar los casos de uso, según se mostrará en el software por construir, y permite eliminar muchos posibles malos entendidos. Cuando se diseñan las interfaces gráficas de usuario, es esencial involucrar a los interesados, y reflejar la visión lógica del sistema a través de las interfaces; por ello, debe haber consistencia entre los modelos conceptuales elaborados por los analistas y el comportamiento real del software por construir.

Sin embargo, un aspecto que no ha sido tratado muy frecuentemente es cómo estos diagramas se integran (Bustos, 2002), es decir, cuáles son las relaciones explícitas posibles entre estos diagramas cuando están describiendo un mismo modelo.

En algunas de las herramientas comerciales para el apoyo a las actividades y procesos de la ingeniería de software (denominadas Computer-Aided Software Engineering) se realizan actualmente chequeos de la consistencia interna de los diagramas, pero se realizan pocas revisiones sobre la consistencia entre los diferentes diagramas o artefactos (Chiorean, et al., 2003), la cual debe ser analizada manualmente y de manera exhaustiva por los analistas y diseñadores del proyecto.

Los artefactos definidos por UML deberían guardar consistencia cuando se traten del mismo modelo. La consistencia interna de cada artefacto está definida en la especificación de UML, pero la consistencia entre diferentes artefactos poco se ha trabajado; además, aún subsisten problemas:

- La consistencia intermodelos no se especifica formalmente (Glinz, 2000). Una especifica-

ción formal de este tipo de consistencia facilita la comprensión de la información común que deben poseer los diferentes artefactos y posibilita su posterior implementación en herramientas computacionales.

- Sólo se realizan algunos chequeos de consistencia de manera automática entre alguno de los artefactos y el código resultante (Gryce, et al., 2002). El código del software se elabora en una etapa posterior al análisis y el diseño; si el chequeo de la consistencia se realiza sobre el código del software, es probable que algunas de las inconsistencias previas hayan sobrevivido hasta la fase de implementación y, como se dijo previamente, es preferible encontrar este tipo de defectos en las fases iniciales del desarrollo.
- En algunos trabajos se definen reglas de consistencia con reglas de manera formal pero sólo en el nivel de intramodelo (Chiorean, et al., 2003), (OMG, 2007). Los modelos no se suelen construir con únicamente un diagrama; se requiere la participación de diferentes puntos de vista que suelen ser expresados en diferentes diagramas. Si esos diagramas se deben referir a la misma información, las reglas de consistencia intramodelo tienen un restringido campo de acción y pueden permitir que subsistan errores de consistencia entre los diferentes diagramas.

En este artículo se propone un método para verificar la consistencia entre el diagrama de clases y el diagrama de casos de uso de UML de una manera formal, evaluando una serie de reglas definidas en OCL (OMG, 2007), las cuales se deben cumplir para garantizar que la información brindada por dichos modelos sea consistente. Se define, adicionalmente, la consistencia con las interfaces gráficas de usuario, ya que éstas son construidas con gran participación de ambos diagramas.

Este artículo está organizado de la siguiente manera: en la sección 2 se muestra la problemática general que motiva este trabajo de investigación; en la sección 3 se presenta la revisión de la literatura especializada que se pudo recopilar alrededor del tema en estudio; en la sección 4 se expone el método relacionado con la construcción de las reglas de consistencia y se describe el método para la validación de éstas. Finalmente, en la sección 5 se exponen las conclusiones y el trabajo futuro que se puede derivar de este artículo.

1. PROBLEMÁTICA GENERAL DE INVESTIGACIÓN

Existe gran cantidad de trabajos que abordan el problema de la consistencia en el nivel de intramodelo, tanto de manera formal como informal; sin embargo, son pocos los autores que han trabajado la consistencia entre modelos, la cual tampoco se ha expresado en lenguajes formales. Otros trabajos realizan consistencia entre diagramas y código resultante, lo que implica no poder comprobar reglas de consistencia antes de la implementación y las pruebas del software.

Para el caso específico, la mayoría de los trabajos que tratan el tema de la consistencia entre el modelo de clases y el modelo de casos de uso de UML lo hacen de una manera informal, en la que recurren al procesamiento de lenguaje natural para garantizar la correspondencia y completitud de los elementos de ambos diagramas, utilizando principalmente los escenarios y la descripción de los casos de uso para esto.

Por otro lado, ninguno de los trabajos recopilados tiene en cuenta las interfaces gráficas de usuario para el manejo de la consistencia entre estos modelos, aún a sabiendas de la alta relación que existe entre éstas y ambos diagramas. Por lo tanto, se propone como una solución a algunas de las limitaciones anotadas el planteamiento de un conjunto de reglas de consistencia entre el diagrama de casos de uso y el diagrama de clases de UML que tengan en cuenta las interfaces gráficas

de usuario, así como la definición de una especificación formal de estas reglas de consistencia; de esta manera, además de permitir un chequeo de la consistencia entre los dos diagramas, se puede involucrar la validación que de ellos puedan realizar los interesados por medio de las interfaces gráficas de usuario.

El tema de la consistencia se ha trabajado más en el nivel intramodelo, donde se verifica que todos los elementos de un mismo diagrama o artefacto sean consistentes entre sí, pero sin tener en cuenta las relaciones con elementos de otros artefactos que pertenecen al modelo. En la parte de intermodelos, el trabajo desarrollado en consistencia ha sido relativamente poco, donde se muestran propuestas de métodos aislados para llevar a asegurar consistencia entre diferentes modelos, pero en general no se muestra un mecanismo formal que pueda ser aplicado por medio de un lenguaje de especificación como el OCL. En general, la falta de formalismo en la definición de las reglas puede conducir a problemas de ambigüedad en la interpretación de tales reglas, y finalmente a una implementación errónea de las mismas.

Se necesita, por tanto, una especificación formal de reglas de consistencia entre el modelo de casos de uso y el modelo de clases, empleando un lenguaje de especificación (que en este caso será OCL); estas reglas se podrían aplicar desde las fases de definición y análisis del ciclo de vida del software, donde se tienen versiones preliminares de ambos diagramas, o incluso en fases posteriores de diseño previas a la implementación, donde los diagramas son más refinados por la cantidad de detalles que se deben precisar en ese proceso. Con ello, se busca que los analistas no inviertan demasiado tiempo y dinero en la revisión exhaustiva de los diagramas en busca de su consistencia, y que se detecten los errores potenciales en fases relativamente tempranas del desarrollo.

Las reglas que se planteen deberán tomar en consideración no sólo los errores que se pueden cometer al trabajar con puntos de vista independientes, sino también advertencias de posibles errores que pueden llegar a ser nocivos para el modelamiento que se está realizando del producto software. De esta manera, podría ser posible advertir a los analistas de los errores reales y potenciales que están cometiendo en la generación de los diagramas, como una especie de extensión de las reglas de buena formación de los mismos, pero tomando en consideración la interrelación con otros diagramas.

2. REVISIÓN DE LA LITERATURA

La principal fuente de reglas de consistencia para los diagramas de UML es la especificación misma emanada por el OMG (OMG, 2007). En dicha especificación, se incluyen algunas reglas de consistencia intramodelos, ya sea de los diagramas de casos de uso o de los diagramas de clases, pero no se definen reglas de consistencia intermodelos. Esas reglas intramodelos se encuentran plenamente definidas tanto en lenguaje natural como en OCL y algunas de ellas se encuentran incorporadas en algunas herramientas CASE convencionales, como es el caso de ArgoUML®.

Para el manejo de estas reglas de consistencia entre modelos UML se han trabajado básicamente algunos métodos:

- Xlinkit (Gryce, et al., 2002) es un entorno para chequear la consistencia de documentos heterogéneos distribuidos. En esta propuesta se hace uso de XML (W3C, 2007), Xpath (W3C, 2007), Xlink (W3C, 2007) y DOM (W3C, 2007), y está conformada por un lenguaje basado en lógica de primer orden, que permite expresar restricciones entre documentos, un sistema de manejo de documentos y un motor que chequea las restricciones contra los documentos. Para explicar la operación de Xlinkit, se muestra en un ejemplo el caso de dos desarrolladores trabajando indepen-

dientemente en el mismo sistema, con su información guardada en diferentes máquinas. Uno está especificando un modelo UML y el otro trabajando en una implementación en Java. El objetivo es chequear una restricción simple, que cada clase en el modelo UML tenga que ser implementada como una clase en Java. Esta restricción se debe satisfacer en el modelo y en la implementación para que sean consistentes. Xlinkit provee "conjuntos de documentos" para incluir los documentos y "conjuntos de reglas" para seleccionar las restricciones. La figura 1 muestra el conjunto de documentos para el ejemplo, que consta de un documento de UML (UMLmodel.xml), un documento Java (Main.java) y un documento para la definición de las reglas (ClassSet.xml). En la figura 2 se muestra una restricción descrita en la codificación XML de Xlinkit, que establece que por cada clase que se encuentre en el documento UML debe existir una clase en el documento Java.

```
<DocumentSet name="UMLandJava">
    <Description>
        A UML model and some Java files
    </Description>

    <Document href="http://host1/UMLmodel.xml"/>
    <Document href="http://host2/Main.java" fetcher="JavaFetcher"/>
    <Set href="http://host2/ClassSet.xml"/>
</DocumentSet>
```

Fuente: Gryce, et al., 2002

Figura 1. Ejemplo de conjunto de documentos

```
<consistencyrule id="r1">
    <forall var="c" in="//UML:Class">
        <exists var="j" in="/java/class">
            <equal op1="$c/@name" op2="$j/@name"/>
        </exists>
    </forall>
</consistencyrule>
```

Fuente: Gryce, et al., 2002

Figura 2. Restricción de ejemplo

En tiempo de ejecución, Xlinkit aplica las expresiones XPath en la restricción a todos los documentos del conjunto, y así construye un conjunto de nodos para ser chequeados. La figura 3 muestra 2 hipervínculos en una base de vínculos Xlinkit que ha sido generada de la regla de consistencia. Se han generado "vínculos consistentes" entre elementos consistentes, y "vínculos inconsistentes" entre elementos no consistentes.

Se puede observar que la clase UML ha sido vinculada a la clase Java que conforma, y que la clase UML inconsistente se ha identificado, pero no ha sido vinculada a algo, porque no tiene clase Java correspondiente.

Xlinkit soporta el chequeo de documentos UML contra las reglas de buena formación para los elementos estáticos de UML, sin embargo, no incluye las reglas para los elementos dinámicos del

metamodelo, lo que es necesario para una buena revisión de la consistencia entre diferentes tipos de modelos. Además, las reglas usadas para probar los modelos UML fueron traducidas de las reglas de buena formación del OMG. Ya que esas reglas tienen varias desventajas, los resultados obtenidos en el chequeo de las Reglas de Buena Formación de los modelos UML no son siempre correctos. Los resultados se dan como un reporte, que menciona sólo si una regla fue evaluada a falso o a verdadero. Por lo tanto, esta información no es útil en la identificación de causas de la falla de alguna regla.

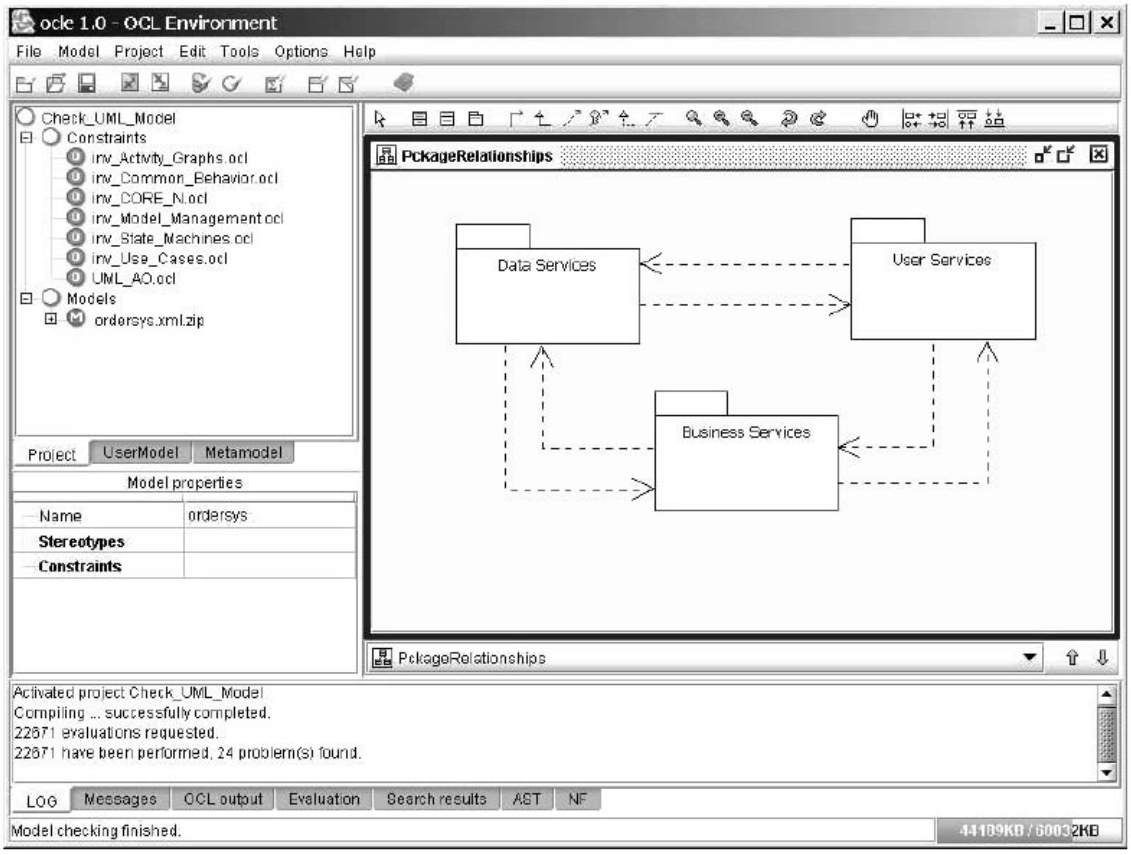
```
<xlinkit:LinkBase docSet="DocSet.xml" ruleSet="RuleSet.xml">
    <xlinkit:ConsistencyLink ruleid="rule.xml#id('r1')">
        <xlinkit:State>consistent</xlinkit:State>
        <xlinkit:Locator xlink:href="http://host1/UMLmodel.xml#//UML:Class[1]"/>
        <xlinkit:Locator xlink:href="http://host2/Main.java#/java/class" fetcher="JavaFetcher"/>
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="rule.xml#id('r2')">
        <xlinkit:State>inconsistent</xlinkit:State>
        <xlinkit:Locator xlink:href="http://host1/UMLmodel.xml#//UML:Class[2]"/>
    </xlinkit:ConsistencyLink>
</xlinkit:LinkBase>
```

Fuente: Gryce, et al., 2002

Figura 3. Hipervínculos resultantes.

En el trabajo de Dan Chiorean (Chiorean, et al., 2003) se trabaja con OCL (Booch, et al., 1997) para chequear la consistencia de modelos UML. Se trabaja con una herramienta CASE UML, Object Constraint Language Environment (OCLE), la cual es totalmente compatible con OCL y soporta nivel de modelo y metamodelo. Esta herramienta es compatible con XMI (OMG, 2007). Puede trabajar con modelos UML generados por las principales herramientas CASE que están disponibles actualmente (Together, Rational Rose, MagicDraw, Poseidon, ArgoUML) (Chiorean, et al., 2003). La herramienta exporta también diagramas UML en formato XMI para que después puedan ser importados y modificados en cualquier herramienta que soporte XMI. OCLE tiene la habilidad de desarrollar diferentes tipos de chequeo de los modelos UML y corregir los errores identificados.

Como ejemplo para chequear la consistencia UML del modelo contra las WFR se trabaja con el "ordersys". Es una aplicación de sistema de pedidos desarrollada para una compañía distribuidora de comida de mar. Este modelo incluye diferentes librerías de Visual Basic. El ejemplo contiene el modelo de aplicación y el proyecto de Visual Basic asociado. El modelo fue exportado en formato XMI desde Rational Rose e importado en OCLE. El proyecto OCLE contiene el modelo UML y el conjunto de reglas OCL guardado en varios archivos:



Fuente: Chiorean, et al., 2003

Figura 4. Explorador de proyectos, panel de salida y un diagrama de clases

Las reglas usadas en el ejemplo son las WFR que definen la semántica estática de UML. Considerando que el proyecto de Visual Basic asociado es ejecutable, se busca ver si la información del diseño del modelo es consistente con la información de la implantación del modelo. El resultado de la primera evaluación se muestra en el panel de salida (figura 4): se encontraron 24 problemas de 22671 evaluaciones realizadas.

La primera regla que se consideró es definida en el contexto de la metaclase Namespace:

```
-- [1] If a contained element, which is not an Association or Generalization
-- has a name, then the name must be unique in the Namespace.

    let noe:  Set(ModelElement) = self.ownedElement->reject(e |
        e.oclIsKindOf(Association) or e.oclIsKindOf(Generalization)) in
    if noe->reject(e | e.name='')->isUnique(e | e.name)
        then true
        else noe->select(e | noe->exists(ae | ae <> e and e.name =
            ae.name))->sortedBy (e | e.name)->isEmpty
    endif
```

Fuente: Chiorean, et al., 2003

Una traducción de la especificación informal es más simple:

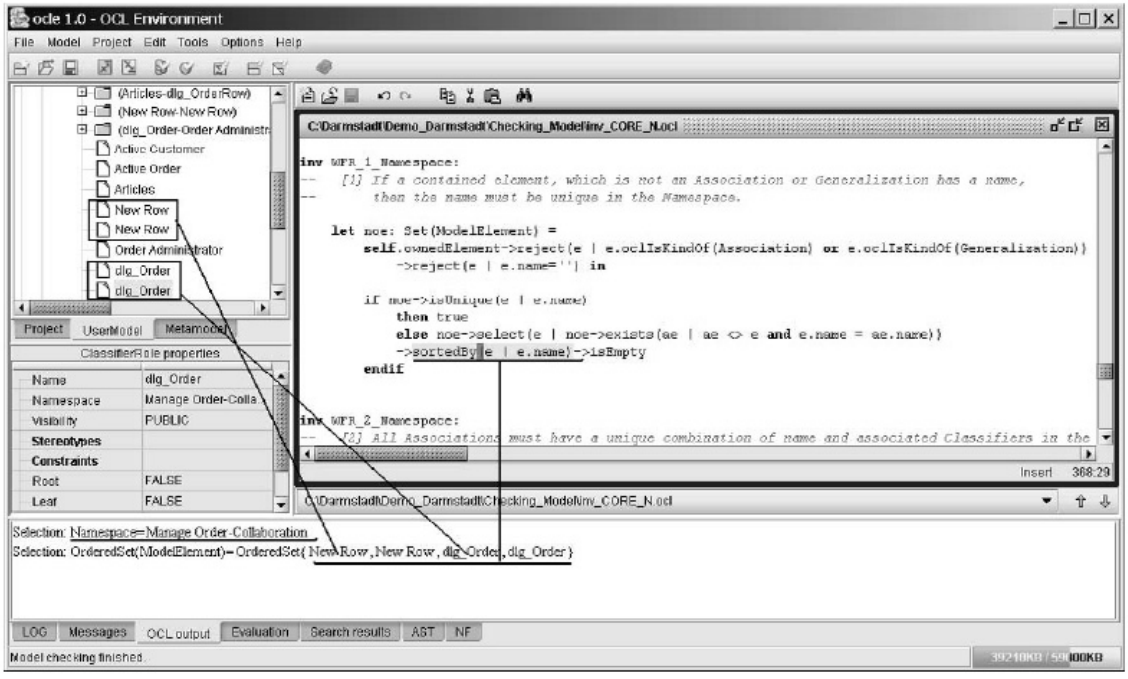
```
self.ownedElement->reject(e | e.oclIsKindOf(Association) or
e.oclIsKindOf(Generalization))->isUnique(e | e.name)
```

Fuente: Chiorean, et al., 2003

Los elementos del modelo sin nombre fueron rechazados porque, aparte de las asociaciones y generalizaciones, hay otros elementos del modelo (como dependencias, instancias, etc.) cuyos nombres no se pueden definir en la mayoría de las herramientas CASE actuales (Chiorean, et al., 2003).

Los elementos que causaron la falla fueron cuatro roles clasificadores, dos de ellos llamados NewRow y los otros llamados dlg_Order (ver Figura 5). Cambiar los nombres de esos elementos resolverá el problema.

Los resultados obtenidos en este y otros ejemplos (Chiorean, et al., 2003) demuestran que usar OCL en el chequeo de reglas de consistencia del modelo UML es un muy buen acercamiento, ya que define todo lo concerniente a las reglas de consistencia de modelos UML en el nivel del metamodelo, por lo que esas reglas son independientes del modelador, soportando su reutilización en cualquier modelo UML. Esta aproximación sólo soporta la validación de la semántica de diagramas individuales de UML, manejando así la consistencia en el nivel de intramodelos.



Fuente: Chiorean, et al., 2003

Figura 5. Dos conflictos de nombre en el Namespace.

En los acercamientos que tratan la consistencia específicamente entre el diagrama de casos de uso y el diagrama de clases se encuentran los siguientes trabajos:

En el trabajo de Kösters, et al. (1997) se asocian el diagrama de casos de uso y el diagrama de clases al derivar el modelo de clases de los casos de uso, y anotando una especificación gráfica de casos de uso con los nombres de las clases que implementan ese caso de uso. Esta es una vista orientada al diseño donde el usuario procede del modelo de casos de uso al modelo de clases, y del modelo de clases a la implementación; sin embargo, no es una aproximación formal.

Un trabajo similar se encuentra en el trabajo de Liu et al (2003), quienes automatizan el proceso de generación del diagrama de clases tomando como punto de partida las especificaciones textuales de los casos de uso, definiendo para ello unas plantillas especiales que capturan la información. En este caso, no se define la consistencia entre los dos tipos de diagrama, sino que se usa una descripción en un lenguaje controlado y de allí se obtiene el diagrama de clases.

De manera análoga, Shiskov et al (2002) procuran el mismo fin, pero, partiendo de lenguaje natural, generan el diagrama de clases utilizando como punto intermedio el diagrama de casos de uso. Para lograr ese fin, emplean los denominados análisis de normas, un conjunto de reglas heurísticas que examinan plantillas de información para transferirlas a los dos diagramas. Por el hecho de generar ambos diagramas desde la misma especificación textual, la consistencia queda garantizada, pero no se definen las reglas de consistencia que deberían existir entre los dos diagramas una vez se modifiquen.

El trabajo de Glinz (2000) también es similar, pues trabaja el diagrama de casos de uso y el diagrama de clases como complementos de una misma especificación de requisitos, lo cual ninguno de los acercamientos mencionados lo hace; sin embargo, el punto de partida es, nuevamente, una especificación textual de los casos de uso en un formato especial. Además, el análisis que se realiza requiere una alta participación del analista en el proceso de integración de ambos artefactos, lo cual hace que su automatización se dificulte.

Buhr (1998) mira los casos de uso como caminos causales que van hacia un modelo de objetos jerárquico. Los puntos de responsabilidad unen segmentos de un camino a elementos del modelo objetual. El modelo objetual y los caminos del diagrama de casos de uso se visualizan juntos en los llamados mapeos de casos de uso. La aproximación de Buhr es orientada al diseño también. Además el camino a través del modelo objetual es todo lo que se conoce del caso de uso; no hay especificación independiente de los casos de uso. Así, la importancia de la separación de lo orientado al usuario se pierde, lo que es una de las fortalezas de la especificación basada en casos de uso. A diferencia de Kösters, et al (1997), quienes se enfocan en la transición de los casos de uso al diagrama de clases, Buhr (1998) usa los casos de uso para visualizar la dinámica del modelo objetual.

El trabajo de Grundy, et al (1998) también maneja la consistencia intermodelos, pero difiere del de Glinz (2000) en que se enfocan en herramientas de entorno de desarrollo de software, manejando inconsistencias en múltiples vistas que son derivadas de un repositorio común. Los artefactos en el repositorio se representan formalmente por una clase especial de gráfico, permitiendo así detección automática de inconsistencia.

Sunetnanta y Finkelstein (2003) presentan un enfoque para el chequeo de consistencia intermodelos basado en la conversión de los diferentes diagramas en grafos conceptuales y la definición de las reglas de consistencia en estos mismos grafos. Los grafos conceptuales no pueden ser considerados como un enfoque formal para la elaboración de este tipo de especificaciones, sino más bien un enfoque semiformal. Además, definen reglas entre

los diagramas de casos de uso y colaboración (el de la versión 1.4 de UML, que actualmente se denomina diagrama de comunicación en la versión 2.0), y no definen las reglas de consistencia entre diagramas estructurales y dinámicos.

Un aspecto a considerar es que todos estos trabajos no incluyen las interfaces de usuario como medio complementario y necesario para validar la consistencia entre dichos modelos, sabiendo que hay una alta correlación entre el modelo de casos de uso e Interfaces, ya que el modelo de casos de uso está motivado y enfocado principalmente hacia los sistemas de información, donde los usuarios juegan un papel primordial, por lo que es importante relacionarse con las interfaces a ser diseñadas en el sistema (Weitzenfeld, 2005).

El trabajo de Glinz (2000) se diferencia específicamente del planteado en este artículo en que no hace uso de un lenguaje formal para generar reglas de consistencia, sino que lo hace de modo informal al completar la documentación de los casos de uso con el mayor detalle posible y, como se mencionó antes, tampoco hace uso de las interfaces para tratar el problema de la consistencia, aspecto que debe ser tenido en cuenta, dado que estas interfaces sirven para apoyar de mejor manera la descripción de los casos de uso además de servir de base para prototipos iniciales.

3. MÉTODO PROPUESTO

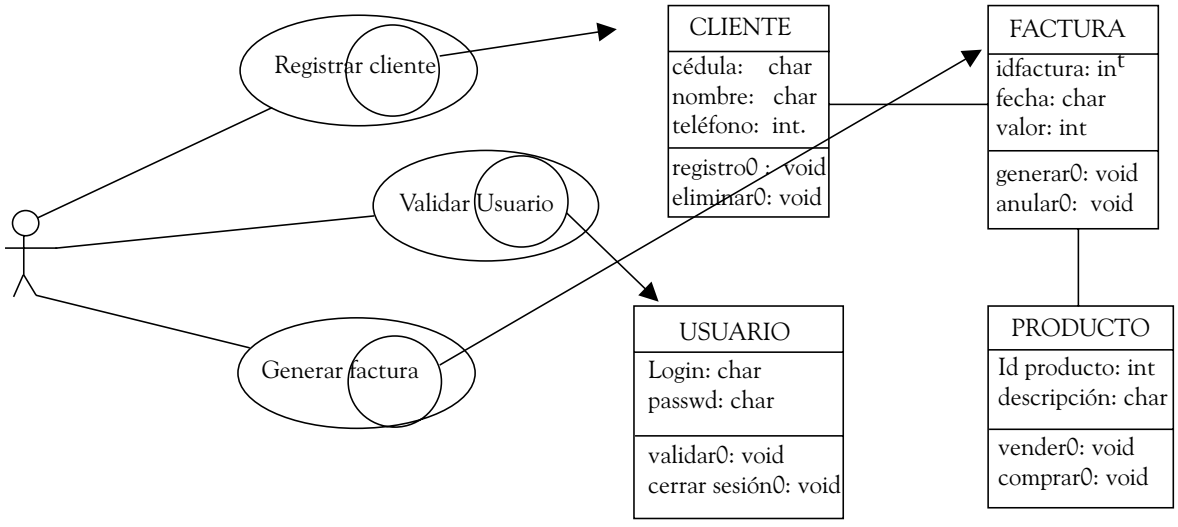
Para definir las reglas de consistencia entre el diagrama de casos de uso y el diagrama de clases UML, se debe definir también el alcance del problema particular, que es tratado bajo las siguientes premisas:

- Cada clase se distingue por su nombre, un conjunto de atributos o propiedades y un conjunto de operaciones ofrecidas por la clase.
- El modelo de casos de uso consta de actores que representan los roles que los diferentes usuarios pueden desempeñar, y los casos de uso que representan lo que deben estar en capacidad de hacer los usuarios con el software por construir. Otro término importante son los escenarios, que corresponden a secuencias específicas de acciones e interacciones entre los actores y el sistema; los escenarios, sin embargo, no serán tenidos en cuenta para esta propuesta, debido a que son una especificación no formal de los pasos llevados a cabo en cada caso de uso y tendrían que ser tratados por procesamiento de lenguaje natural, por lo que están fuera del alcance de este artículo, en el cual se busca proponer una especificación formal.
- El modelo de interfaces especifica cómo interactúa el software por construir con actores externos al ejecutar los casos de uso; en particular, en los sistemas de información ricos en interacción con el usuario, especifica cómo se visualizarán las interfaces gráficas de usuario y qué funcionalidad ofrecerá cada una de ellas (Weitzenfeld, 2005). Para el trabajo propuesto, una interfaz común consta de un título, etiquetas, campos de texto, campos de selección, uno o varios botones de enviar (submit) y un botón de cancelar o salir. Se suponen interfaces sencillas que sólo involucran elementos de una clase.

Para la elaboración de este trabajo, se toma en consideración la relación existente entre el diagrama de clases (OMG, 2007), el diagrama de casos de uso (Jacobson, 1992) y las interfaces gráficas de usuario (Weitzenfeld, 2005). Además, se considera el análisis heurístico que se puede obtener a partir de la experiencia por parte de analistas de software y, tomando como base la estructura de los metamodelos de esos diagramas (OMG, 2007) se proponen 8 reglas de consistencia, que se presentan seguidamente.

Regla 1

El nombre de un caso de uso debe incluir un verbo y un sustantivo. El sustantivo debe corresponder al nombre de una clase del diagrama de clases. En otras palabras, para todo caso de uso U del diagrama de casos de uso, debe existir una clase C perteneciente al diagrama de clases, tal que U.name contenga a C.name. La expresión gráfica de esta regla se puede apreciar en la figura 6.



Fuente: los autores

Figura 6. Descripción gráfica de la regla 1.

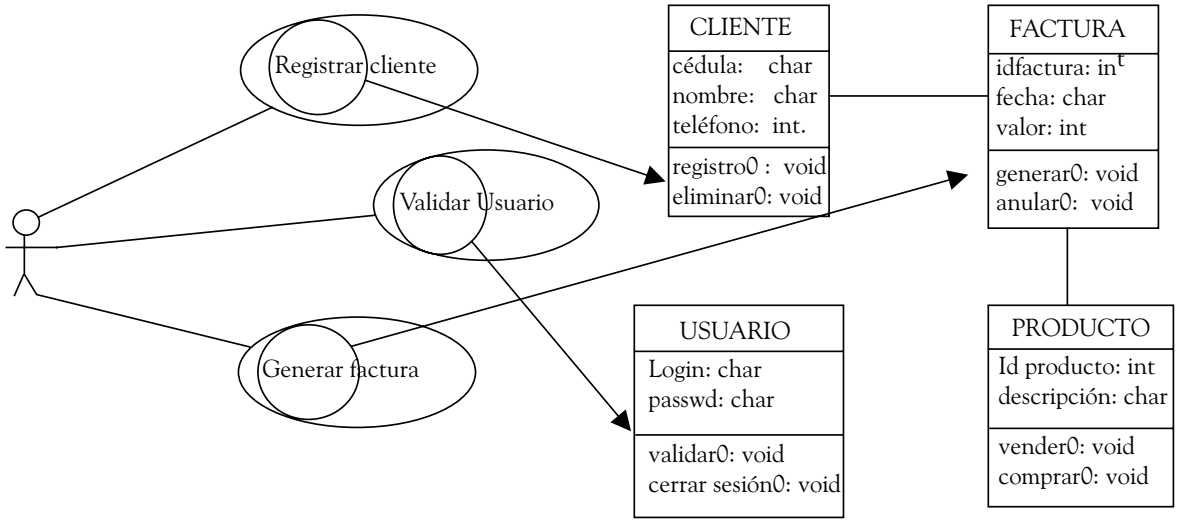
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.UseCase->exists(us: Usecase, c: Class, x: Integer, y: Integer | y > x and us.name.toUpper.substring(x,y)=c.name.toUpper)

Regla 2

El nombre de un caso de uso debe incluir un verbo y un sustantivo. El verbo debe corresponder a una operación de una clase del diagrama de clases que se identificó en la regla 1. En otras palabras, para todo caso de uso U debe existir una clase C que contenga una operación Operationx tal que U.name contenga a C.Operationx. La expresión gráfica de esta regla se puede apreciar en la figura 7.



Fuente: los autores

Figura 7. Descripción gráfica de la regla 2.

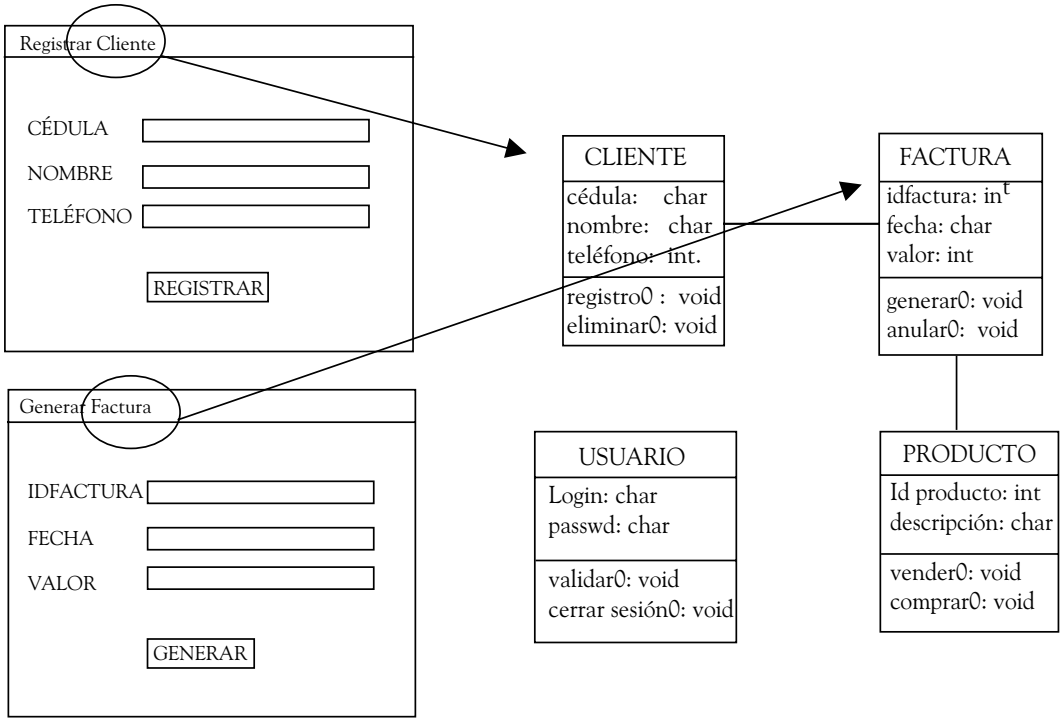
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.UseCase->exists(us: Usecase, c: Class, x: Integer, y: Integer | y > x and us.name.toUpper.substring(x,y)=c.operation.toUpper)

Regla 3

En el título de cada interfaz gráfica de usuario debe ir un verbo y un sustantivo. El sustantivo debe corresponder al nombre de una clase que se encuentre en el diagrama de clases. Dicho de otra forma, para toda interfaz de usuario I debe existir una clase C tal que I.key=1 (título) y que I.value contenga a C.name. La expresión gráfica de esta regla se puede apreciar en la figura 8.



Fuente: los autores

Figura 8. Descripción gráfica de la regla 3.

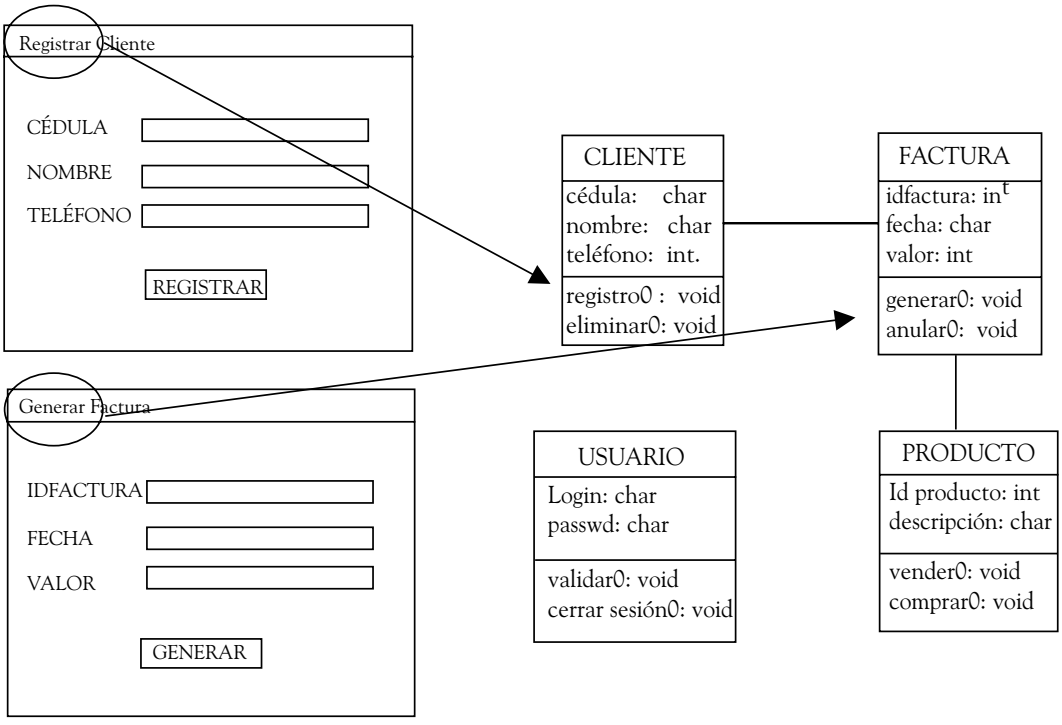
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.DiagramElement->exists(de: DiagramElement, c: Class, x: Integer, y: Integer
| y > x and de.key=1 and de.value.toUpper.substring(x,y)=c.name.toUpper)

Regla 4

Una interfaz gráfica de usuario tiene en su título un verbo y un sustantivo. Dicho verbo debe corresponder a una operación de la clase identificada en la regla 4; dicho de otra forma, para toda interfaz de usuario I debe existir una clase C que contenga una operación Operationx en la que I.key=1 (título) y que I.value contenga a C.Operationx .La expresión gráfica de esta regla se puede apreciar en la Figura 9.



Fuente: los autores

Figura 9. Descripción gráfica de la regla 4.

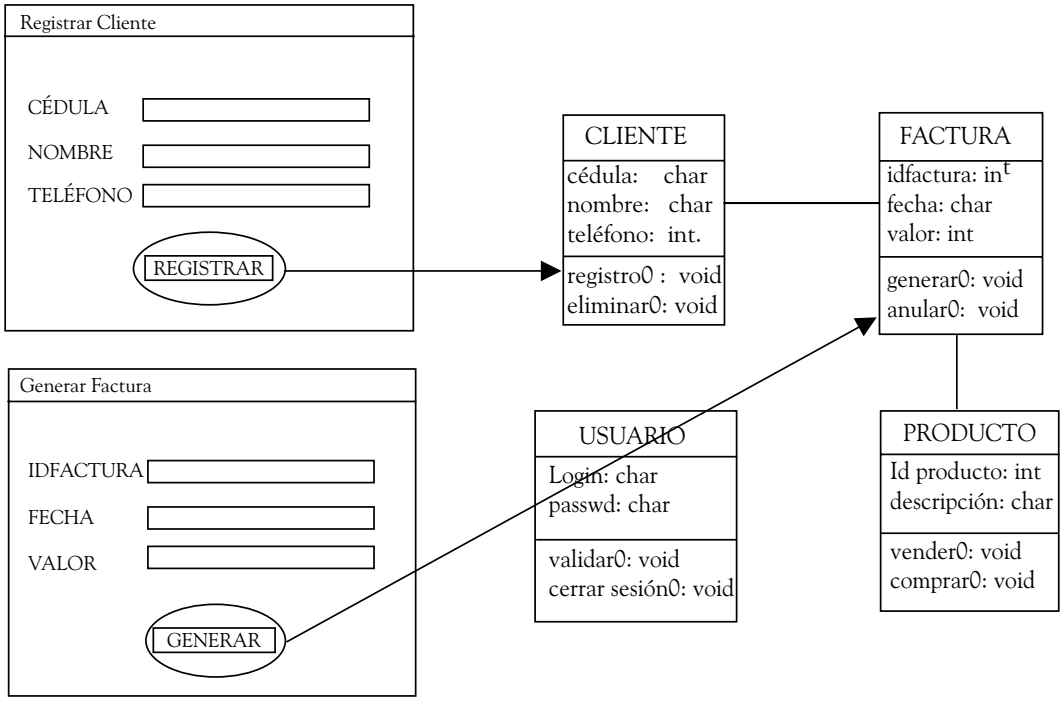
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.DiagramElement->exists(de: DiagramElement, c: Class, x: Integer, y: Integer
| y > x and de.key=1 and de.value.toUpper.substring(x,y)=c.operation.toUpper)

Regla 5

En una interfaz gráfica de usuario debe existir un formulario con un botón para aplicar cambios o enviar información a otro formulario. Dicho botón tiene en su etiqueta un verbo. Dicho verbo debe corresponder a una operación de una clase. Dicho de otra manera, para toda interfaz de usuario I en la que exista un botón de enviar (submit) B debe existir una clase C que contenga una operación Operationx tal que I.key=3 (button) y I.value (label) contenga a C.Operationx. La expresión gráfica de esta regla se puede apreciar en la figura 10.



Fuente: los autores

Figura 10. Descripción gráfica de la regla 5.

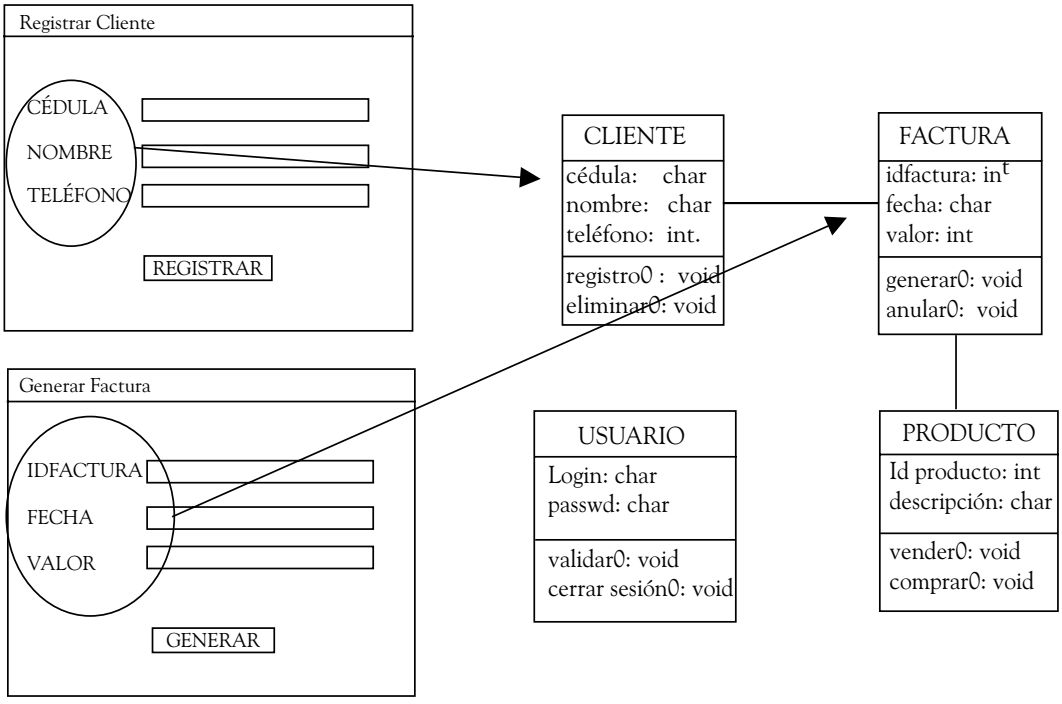
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.DiagramElement->exists(de: DiagramElement, c: Class, x: Integer, y: Integer
| y > x and de.key=3 and de.value.toUpper.substring(x,y)=c.operation.toUpper)

Regla 6

Si una interfaz gráfica de usuario posee campos de texto para que el usuario ingrese información, estos campos deben ir precedidos por sus respectivas etiquetas, las cuales informan acerca de lo que se debe digitar en los campos. Dichas etiquetas deben tener sus atributos correspondientes en una clase. Dicho de otra forma, para toda interfaz de usuario I en la que existan etiquetas (labels) E1, E2, E3,...,Eq con sus respectivos campos de texto T1,T2,T3,...,To y/o campos de selección S1,S2,S3,...,Sp debe existir una clase C con atributos A1, A2, A3,...,An, que contengan a las etiquetas Ex. La expresión gráfica de esta regla se puede apreciar en la figura 11.



Fuente: los autores

Figura 11. Descripción gráfica de la regla 6.

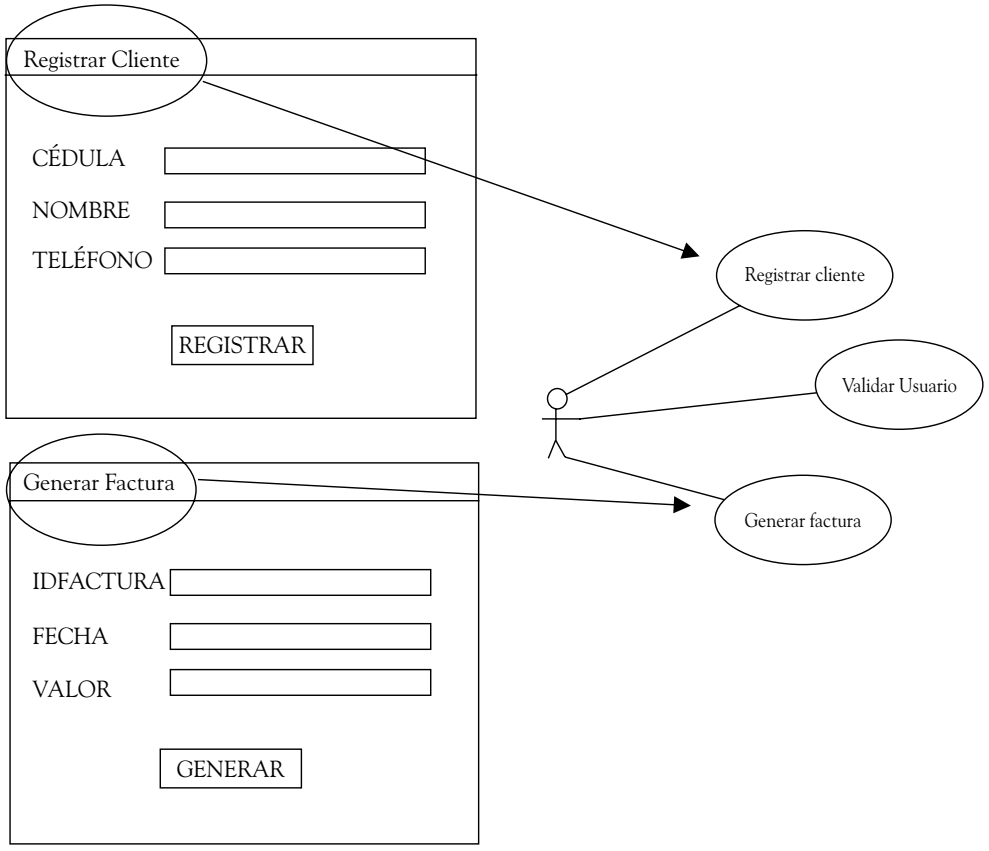
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.DiagramElement->forall(de: DiagramElement, c: Class, x: Integer, y: Integer | de->exists(de.key=3) and (de.key=4 or de.key=5) implies c.atributte->includes(de.value))

Regla 7

En el título de cada interfaz gráfica de usuario debe ir un verbo y un sustantivo. Dicho verbo y sustantivo deben corresponder con el nombre de un caso de uso, los cuales también están compuestos de un nombre y un sustantivo. En otras palabras, para toda interfaz de usuario I debe existir un caso de uso U en el que I.key=1(título) y que I.value = U.name. La expresión gráfica de esta regla se puede apreciar en la Figura 12.



Fuente: los autores

Figura 12. Descripción gráfica de la regla 7.

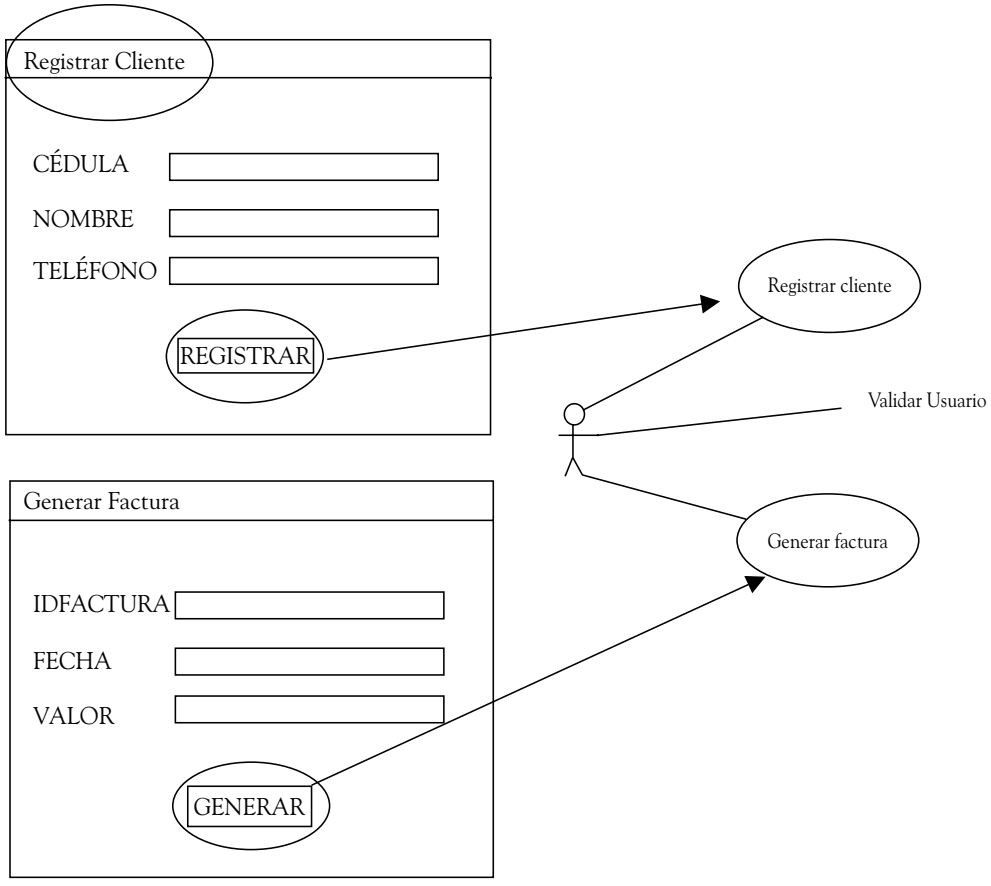
La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.DiagramElement->exists(de: DiagramElement, uc: UseCase | de.key=1 and de.value.toUpper=uc.name.toUpper)

Regla 8

En una interfaz gráfica de usuario debe existir un formulario con un botón para aplicar cambios o enviar información a otro formulario. Dicho botón tiene en su etiqueta un verbo. Dicho verbo debe corresponder a un verbo de un nombre de un caso de uso. Dicho de otra forma, para toda interfaz de usuario I en la que exista un botón de enviar (submit) B (I.key=3) debe existir un caso de uso U tal que U.name contenga a I.value. La expresión gráfica de esta regla se puede apreciar en la Figura 13.



Fuente: los autores

Figura 13. Descripción gráfica de la regla 8.

La expresión en OCL que representa esta regla es la siguiente:

Classifier

self.DiagramElement->exists(de: DiagramElement, uc: UseCase, x: Integer, y: Integer | y > x and de.key=3 and uc.name.toUpper.substring(x,y)=de.value.toUpper)

Para la validación de las reglas mencionadas anteriormente, se hizo uso de dos herramientas que poseen la utilidad de exportar los contenidos de sus modelos en formato XMI (OMG, 2007): ArgoUML® y Visio®. Por estar ArgoUML® enfocado al manejo de diagramas UML, se trabajaron en esta herramienta los diagramas de casos de uso y de clases; en Visio® se trabajaron las Interfaces Gráficas de Usuario (GUI).

Después de hacer los respectivos modelos e interfaces, estos se exportan en formato XMI y luego se analizan con una herramienta que se desarrolló para efectos de este trabajo en el lenguaje de programación Java®. Esta herramienta hace uso del lenguaje Xquery por medio de una aplicación especializada denominada Saxonb para la validación de las reglas. En este programa se evalúan las diferentes reglas con los tres modelos y para cada regla sale un informe que indica si la regla se cumple (estado correcto), si no se cumple (estado de error) o si posiblemente hay un error de sinónimos por lo que se presenta una advertencia (warning).

Con el fin de tener la posibilidad de mostrarle al usuario que posiblemente algunas de las inconsistencias encontradas se debieron al uso de sinónimos por parte del analista y que no son errores como el caso de que faltara una clase o un caso de uso, se da la posibilidad de mostrar advertencias donde se indique que el posible error puede ser debido a uso de palabras similares. Se implementó una lista de los sustantivos y verbos más utilizados en el ámbito del área de software, teniendo como base los entregables de varios semestres presentados por los estudiantes en la asignatura Ingeniería de Software de la Universidad Nacional.

4. CONCLUSIONES

Los resultados obtenidos en las diferentes áreas de trabajo de la investigación muestran un número considerable de inconsistencias que pueden ser encontradas con el método propuesto, debido que al incluir las interfaces gráficas de usuario se tiene un apoyo extra en la verificación de correspondencia de sus elementos con los atributos de las clases, garantizando así una mayor consistencia entre los modelos de casos de uso y de diagramas de clases de UML.

Por otro lado, al presentar una especificación formal de reglas de consistencia entre el diagrama de casos de uso y diagrama de clases en OCL, se formaliza la validación de los aspectos necesarios para garantizar que no exista ambigüedad entre estos modelos y que no se tenga diagramas con objetos aislados. Al integrar las interfaces de usuario con estos dos modelos utilizando archivos en formato XMI, se hace al método independiente de la plataforma y que sea regido por el estándar, asegurando así interoperabilidad y modularización para facilitar el intercambio de información.

A diferencia de otros trabajos, en este paper se planteó un conjunto de reglas de consistencia intermodelos, específicamente entre el diagrama de casos de uso y el diagrama de clases. Además de definir dicho conjunto de reglas, se implementó un método que permite validar dichas reglas y que puede ser utilizado para revisar otras reglas, brindando la posibilidad de extender el método a otros modelos.

La integración del lenguaje de especificación de objetos, OCL, en el método propuesto es un aspecto a tener en cuenta debido a que no se encontraron evidencias de que algún autor haya presentado reglas de consistencia entre el modelo de clases y el modelo de casos de uso de UML de manera formal, aportando así un método de validación de reglas de consistencia sin ambigüedades y bien formulado. Esto implica una posible integración con las reglas de buena formación en la especificación de UML, definiendo así una posibilidad de integrarse en el estándar.

Al formalizar las reglas se elimina la posibilidad de darle varias interpretaciones y se descarta el

manejo de lenguaje natural, estableciendo así un mecanismo consistente y concreto para la verificación de consistencia entre dichos modelos.

Ninguna investigación que trabaja la consistencia entre el diagrama de clases y el modelo de casos de uso de UML ha integrado las interfaces de usuario en sus reglas de consistencia, dejando así un gran vacío en los prototipos de usuario para los casos de uso. Al integrar las interfaces de usuario en el método propuesto, se logró validar la correspondencia de los atributos de las clases con los casos de uso, haciéndolo formalmente y pudiendo así integrar de una manera completa las reglas para la validación de la consistencia entre los modelos mencionados previamente. También se compararon los elementos de las interfaces gráficas de usuario con los casos de uso para garantizar una correspondencia en el funcionamiento del sistema tanto en los prototipos como en los casos de uso iniciales, dejando la posibilidad de encontrar situaciones específicas sin modelar o sin su correspondiente prototipo.

Como trabajo futuro se busca extender el método a otras parejas de modelos, con el fin de que exista más consistencia en los diagramas UML que genere el analista, garantizando así un buen producto de software. Se piensa inicialmente integrar el método con el diagrama de actividades y el diagrama de secuencias, debido a su gran relación y afinidad con los diagramas presentados.

REFERENCIAS

BOOCH G., JACOBSON I., and RUMBAUGH J., 1997. "Object Constraint Language Specification", UML Documentation Set Version 1.1, Rational Software Cooperation, September.

BUHR, R.J.A., 1998 Use Case Maps as Architectural Entities for Complex Systems. IEEE Transactions on Software Engineering 24, 12 (Dec. 1998). 1131-1155.

BUSTOS, G., 2002. Integración Informal De Modelos En Uml. Publicado en la Revista Ingenerare Nº 14, Facultad de Ingeniería, Pontificia Universidad Católica de Valparaíso, Valparaíso (Chile), Diciembre 2002.

CHIOREAN, D., PASCA, M., CARCU, A., BOTIZA, C., and MOLDOVAN, S., 2003. Ensuring UML models consistency using the OCL Environment. Sixth International Conference on the Unified Modelling Language - the Language and its applications, San Francisco.

EGYED, A., 2001. Scalable Consistency Checking between Diagrams – The ViewIntegra Approach. En: 16th IEEE International Conference on Automated Software Engineering.

GLINZ, M., 2000. A lightweight approach to consistency of Scenarios and Class Models. En: Fourth International Conference on Requirements Engineering, Illinois (USA), June 10-23.

GRUNDY, J., HOSKING, J., MUGRIDGE, W.B., 1998 Inconsistency Management for Multiple-View Software Development Environments. IEEE Transactions on Software Engineering 24, 11 (Nov. 1998). 960-981.

GRYCE, C., FINKELSTEIN, A. and NENTWICH, C., 2002. Lightweight Checking for UML Based Software Development. En: Workshop on Consistency Problems in UML-based Software Development., Dresden, Germany.

JACKSON M., 1995. "Software Requirements & Specifications: a lexicon of practice, principles and prejudices", Addison Wesley, Great Britain.

JACOBSON, I., 1992 Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley,

JACOBSON, I., BOOCH, I. and RUMBAUGH J. G., 1999. The Unified Software Development Process, Addison Wesley.

KÖSTERS, G., PAGEL, B.-U., WINTER, M., 1997 Coupling Use Cases and Class Models. En: BCS FACS/EROS Workshop on Making Object-oriented Methods more Rigorous. London.

LIU, D., SUBRAMANIAM, K., FAR, B.H., EBERLEIN, A., 2003. Automating transition from use-cases to class model. Canadian Conference on Electrical and Computer Engineering. IEEE CCECE 2003. Pp. 831-834 vol.2

Meta Object Facility (MOF) Specification. OMG Document: formal/2002-04-03, 2003.

OCL. Object Constraint Language Specification. Disponible en Web en: http://www.omg.org/technology/documents/formal/ocl.htm

OMG – Object Management Group, 2007. Disponible vía Web en http://www.omg.org

SUNETNANTA, T. y FINKELSTEIN, A., 2003. Automated Consistency Checking for Multiperspective Software Specifications. Proceedings of the 26th Australasian computer science conference, Volume 16, Pp. 291-300.

Unified Modeling Language: Superestructure versión 2.0. OMG Final Adopted Specification, 2002.

W3C - World Wide Web Consortium., 2007. Disponible via web en: http://www.w3.org/

WEITZENFELD, A., 2005. Ingeniería de Software orientada a objetos con Uml, Java e Internet. Thomson editores.

XMI – XML Metadata Interchanger. Disponible vía Web en http://www.omg.org/technology/xml/

XML - Extensible Markup Language. Disponible vía web en http://www.w3.org/XML/

ZOWGHI, D. and GERVASI, V, 2002. The Three Cs of requirements: consistency, completeness, and correctness. En: International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage, 2002, pp. 155-164.